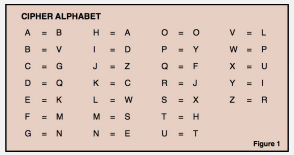
Solitaire

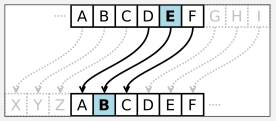
- this assignment is about encryption and secret messages – it has nothing to do with the card game Solitaire
  - the name “Solitaire” comes from the use of a deck of cards in carrying out the encryption scheme, not from any relationship with the game
- while you would use a standard deck of cards to carry out the encryption by hand, this assignment does not use the `Deck` class we’ve used elsewhere
  - you’ll be writing a `SolitaireDeck` class which contains the specific deck-manipulation operations needed for Solitaire encryption

Shift Ciphers

- in a *substitution cipher*, each letter in the original plaintext message is replaced by another letter in the ciphertext as defined by a *cipher alphabet*
  - to decrypt, reverse the substitution
  - both sender and receiver need to know the cipher alphabet (and it must be a secret known only to them)
  - a fixed cipher alphabet is easily broken



- a simple cipher alphabet is a *shift cipher*, where each letter is replaced by a letter shifted a fixed number of places
  - only a single number needs to be known to convey the cipher alphabet
  - only 25 possible shifts means this is very easily broken

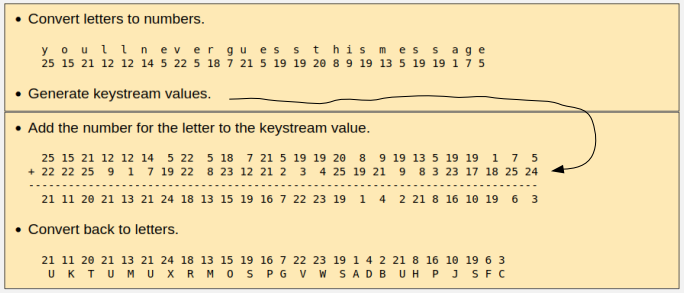


- Solitaire uses a different shift for every character in the plaintext, and uses an easily-memorized algorithm involving a deck of cards and a passphrase to be able to generate the sequence of shifts
  - little information needs to be shared to convey the cipher alphabet (the algorithm, which can be publicly known, and the passphrase)
  - no fixed cipher alphabet – much more secure

https://en.wikipedia.org/wiki/Caesar_cipher
http://accordingtobenedict.com/film-tv/enigma-part-3-the-substitution-cipher/

Solitaire

- the keystream values define the shift applied to each letter in the plaintext, not the cipher alphabet
  - there is not a fixed cipher alphabet – the same letter appearing in different places in the plaintext will encrypt to different ciphertext letters



Solitaire

- breaking things down into modules is essential
  - write `SolitaireDeck`, implementing the deck manipulations used in the encryption scheme
  - write `KeystreamGenerator`, which uses those deck manipulations as building blocks for generating the keystream values needed for encryption/decryption
    - “keying the deck” involves setting up the initial order of cards in the deck
      - necessary so the receiver can duplicate the sender’s sequence for decryption
      - different initial orders provide different sequences
  - write `SolitaireEncoder`, which uses the keystream values to do the actual encryption/decryption

Keying the Deck

- all of the linked list manipulation is done by `SolitaireDeck` – keying the deck just involves calling the right methods

1. Start with the cards of the deck arranged in order from the lowest card to the highest card (and joker A before joker B).
2. Repeat the following steps for each character of the passphrase:
   - If the character is not a letter (e.g. space, punctuation), skip it.
   - Do a joker swap with joker A.
   - Do a joker swap with joker B.
   - Do a triple cut.
   - Do a count cut, using the value of the card at the bottom of the deck as the number of cards to count.
   - Convert the current letter of the passphrase to a number. (A=1, B=2, C=3, etc) Treat capital and lowercase letters as equivalent (so both 'A' and 'a' convert to 1, etc).
   - Do a second count cut, using the number from the previous step as the number of cards to count.

this is what the `SolitaireDeck` constructor does

the passphrase is a string – see the `String` class API for how to get each character

look for a convenient method in the `Character` class

just call the right methods of `SolitaireDeck`

just call the right methods of `SolitaireDeck`

can be a big 'if' or a clever formula – either way, create a private helper method

Keystream Generation

- all of the linked list manipulation is done by `SolitaireDeck` – keystream generation just involves calling the right methods

The central part of the Solitaire algorithm is the generation of a *keystream* - a sequence of numeric values which will then be used to encrypt or decrypt a message. To generate the next value in the keystream:

- Do a joker swap with joker A.
- Do a joker swap with joker B.
- Do a triple cut.
- Do a count cut, using the value of the card at the bottom of the deck as the number of cards to count.
- Look at the top card's value and count down that many cards from the top of the deck, counting the top card as 0. If the resulting card isn't a joker, return that card's value as the next value in the keystream. If that card is a joker, go back to step 1 and repeat the whole process until you get to a card that isn't a joker.

for all of this, just call the right methods of `SolitaireDeck`

Encryption

- all of the keystream-related functionality is done by `KeystreamGenerator` – encryption just involves calling the right methods
- `SolitaireDeck` is not used directly

1. Initialize the keystream generator (see "Keying the Deck", below).
2. For each character in the original message:
   - If the character is not a letter (e.g. a space, punctuation, a number), skip it.
   - Convert the letter to a number. (A=1, B=2, C=3, etc) Treat capital and lowercase letters as equivalent (so both 'A' and 'a' convert to 1, etc).
   - Generate the next keystream value (see "Keystream Generation", below).
   - Add the number for the letter to the keystream value. If the sum is greater than 26, subtract 26 so you end up with a number between 1 and 26.
   - Convert the number back into a letter. (1=A, 2=B, etc) This is the encrypted letter.

this is what `KeystreamGenerator`'s constructor does

the message is a string – see the `String` class API for how to get each character

look for a convenient method in the `Character` class

just call the right methods of `KeystreamGenerator`

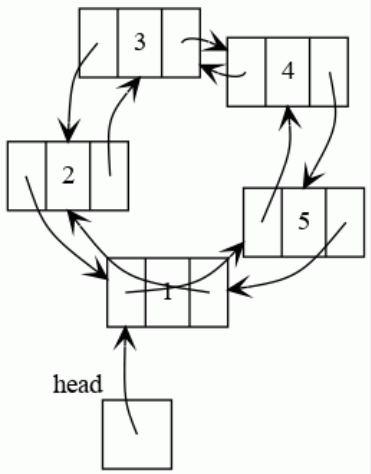
can be a big 'if' or a clever formula – either way, create a private helper method

- decryption is similar

Encryption, Decryption, and the Passphrase

- the deck size depends on the number of possible characters, not the length of the message to encrypt or decrypt
  - a deck size of 26 suffices for encrypting alphabetic characters (a-z; capitals and lowercase letters are treated the same)
    - an extra credit option is to also encrypt numbers, punctuation, and other non-letter characters
  - the deck always has exactly two jokers regardless of the deck size
    - joker value is deck size + 1
- the passphrase is used to put the deck into a reproducible starting order for generating keystream values
  - the passphrase can be any length – it is not connected to the size of the deck or the length of the message to encrypt

Solitaire – Circular Doubly-Linked Lists



- doubly-linked
  - each list node contains prev, next pointers and an element
- circular
  - tail's next points to the head
  - head's prev points to the tail
- no need for tail pointers
  - a tail pointer lets you find the tail without going through the whole list
  - in a circular list, head's prev points to the tail

Solitaire – Circular Doubly-Linked Lists

- `DoubleListNode` class is provided
  - the element stored is a `SolitaireCard`

Constructors

Constructor	Description
`DoubleListNode(SolitaireCard card)`	Create a new node storing the specified card.
`DoubleListNode(SolitaireCard card, DoubleListNode prev, DoubleListNode next)`	Create a new node storing the specified card.

Constructors

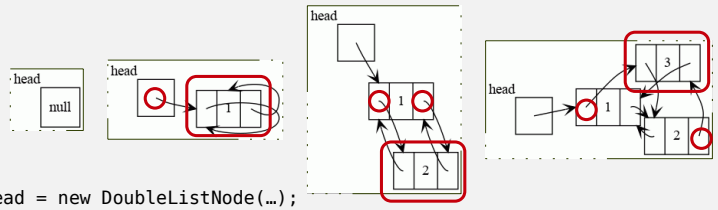
Constructor	Description
`SolitaireCard(int value)`	Create a new (non-joker) card with the specified value.
`SolitaireCard(int value, char joker)`	Create a new joker with the specified values.
`SolitaireCard(java.lang.String str)`	Create a card from its string representation.

Solitaire – Circular Doubly-Linked Lists

- figure out operations involving circular doubly-linked lists the same way as with singly-linked lists
  - draw before and after pictures
  - identify what changes
  - get variables pointing to nodes of interest
  - do the updates
  - consider special cases

Solitaire – Circular Doubly-Linked Lists

- building a circular doubly-linked list
  - if you know the prev and next nodes when you create a new node, you can use the second constructor
  - if not, use the first constructor or the second one with `null` parameters, then use `setNext` and `setPrev`




```
head = new DoubleListNode(…);
head.setNext(head);
head.setPrev(head);
```

```
DoubleListNode newnode =
  new DoubleListNode(…,head,head);
head.setNext(newnode);
head.setPrev(newnode);
```

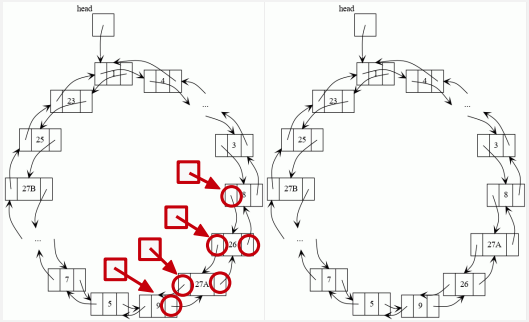
SolitaireDeck

Joker Swap - Joker A

In a joker swap, joker A is swapped with the card that comes after it in the deck. For example: (**bold underline** highli

```
before: 1 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 27A 26 8 3 24 6 4
after:  1 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 8 3 24 6 4
```

- start with the standard case – joker A is not first or last
  - identify what changes
    - rearrange pointers – do not create new nodes
  - get variables pointing to the nodes of interest – joker A, before joker A, after joker A, after after joker A
  - `setNext` and `setPrev` to update links



SolitaireDeck

Joker Swap - Joker A

In a joker swap, joker A is swapped with the card that comes after it in the deck. For example: (**bold underline** highli

```
before: 1 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 27A 26 8 3 24 6 4
after:  1 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 8 3 24 6 4
```

If joker A is last, it is still swapped with the next card (which happens to be the top card when the deck is treated as a change.

```
before: 1 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 4 26 8 3 24 6 27A
after:  1 27A 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 4 26 8 3 24 6
```

This is still a swap, since the sequence of cards surrounding 27A in the "before" deck is ... 24 6 **27A 1** 23 25 ... and ir top card to the bottom of the deck to take the former bottom card's position.

Note that if joker A is first, the top card of the deck *does* change:

```
before: 27A 1 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 8 3 24 6 4
after:  1 27A 23 25 27B 2 22 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 8 3 24 6 4
```

- then trace code for special cases
  - if it breaks, detect and handle the case

SolitaireDeck

Triple Cut

In a triple cut, the cards above the first joker (the one closest to the top of the deck) are exchanged with the cards below the second joker. For this step, it doesn't matter which joker is A or B.

```
before: 1 23 25 2 22 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 8 3 24 6 4
after:  8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 1 23 25 2 22
```


SolitaireDeck

Count Cut

A count cut involves several steps:

- Remove the bottom card from the deck.

```
before: 8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 1 23 25 2 22
after:  8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 1 23 25 2
```

- Count down a specified number of cards from the top of the deck, counting the top card as 1. (How many cards to count is specified as a parameter to this operation.) For example, counting down 22 cards means that the 26 is the current card.
- Do a cut: put the cards you just counted through at the bottom of the deck.

```
before: 8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 27A 1 23 25 2
after:  27A 1 23 25 2 8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26
```

- Add the original bottom card (which you removed) back to the bottom of the deck.

```
before: 27A 1 23 25 2 8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26
after:  27A 1 23 25 2 8 3 24 6 4 27B 10 11 12 13 14 15 16 17 18 19 20 21 7 5 9 26 22
```

Correctness and Testing `SolitaireDeck`

It is easy to make mistakes with linked list manipulations!

- draw before and after pictures to work out the operations
- trace through an example once you've written the code
- identify and check preconditions (for all methods)
- check class invariants related to integrity of the linked list at the end of each method that manipulates the linked list
  - `checkStructure()`, `checkContents()`
- write test cases in `SolitaireTester` as you work on `SolitaireDeck`
  - write test cases after you implement each of the deck operations
    - test all methods that might have bugs – anything more than a simple getter
    - test all cases that might have bugs – all different behaviors, typical special cases
  - `SolitaireDeck` should work with decks of any size, so you can test with a smaller deck (don't have to use 26 cards)